Computer Vision - Tirana May 2023

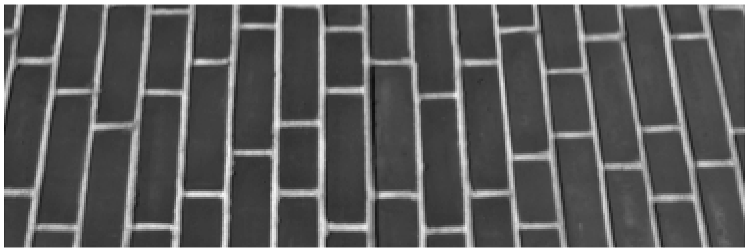
Final Evaluation Exercise: Complete the following exercises

1. Load and visualize the 'bricks' image from the skimage built-in datasets. Report the size of the image and range of the pixel grayscale levels.

```
import matplotlib.pyplot as plt
from skimage import data

# Load the bricks image
image = data.brick()

# Get the size of the image
image_size = image.shape
print("Image size:", image_size)

# Get the range of pixel grayscale levels
pixel_range = (image.min(), image.max())
print("Pixel grayscale range:", pixel_range)

# Visualize the image
plt.imshow(image, cmap='gray')
plt.title("Bricks Image")
plt.axis('off')
plt.show()
```

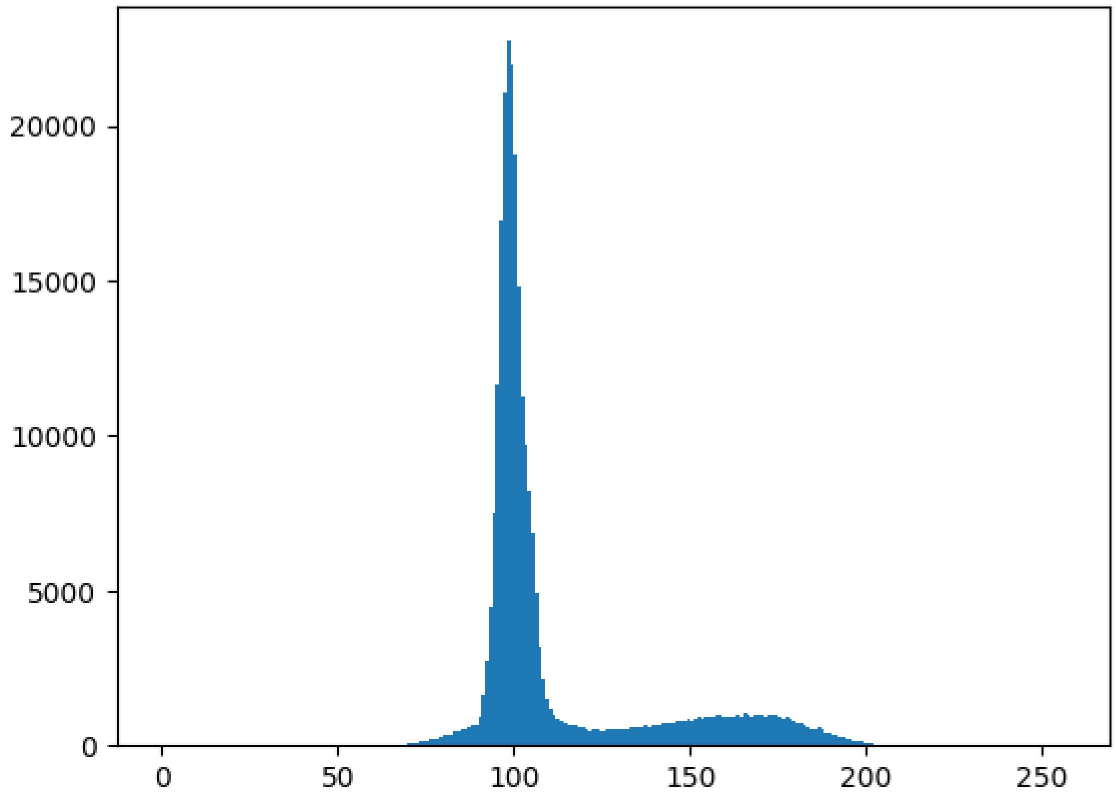
```
Image size: (512, 512)
Pixel grayscale range: (63, 207)
```

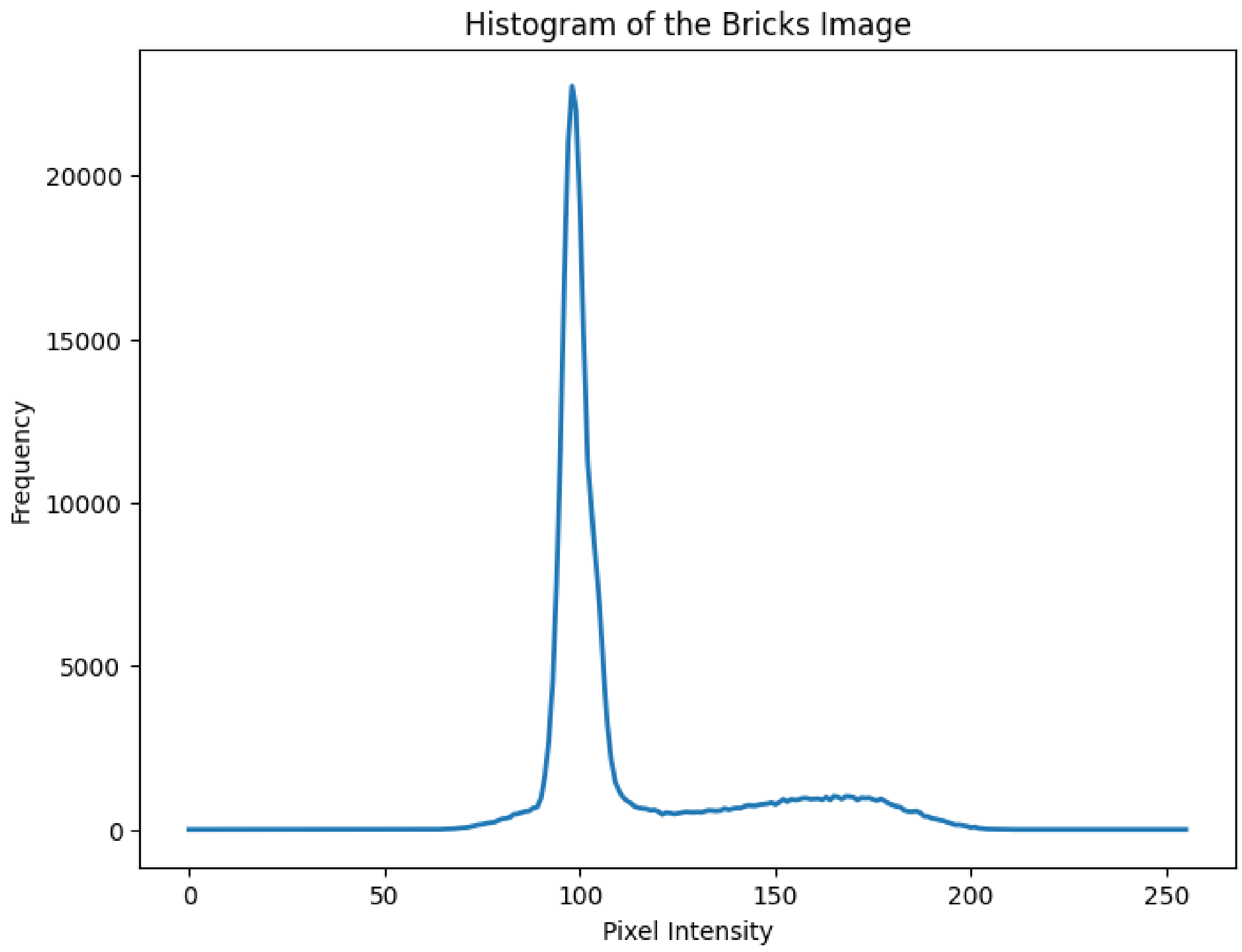
Bricks Image



2. Represent the histogram of the image. Explain the peaks of the histogram in terms of regions of the image.

```
# Compute the histogram
hist, bin_edges, _ = plt.hist(image.ravel(), bins=256, range=(0, 256))

# Plot the histogram
plt.figure(figsize=(8, 6))
plt.plot(hist, lw=2)
plt.title("Histogram of the Bricks Image")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.show()

# Explain the peaks
print("Peaks in the histogram:")
for i in range(1, len(hist) - 1):
    if hist[i] > hist[i-1] and hist[i] > hist[i+1]:
        print("Peak at intensity:", i)
```





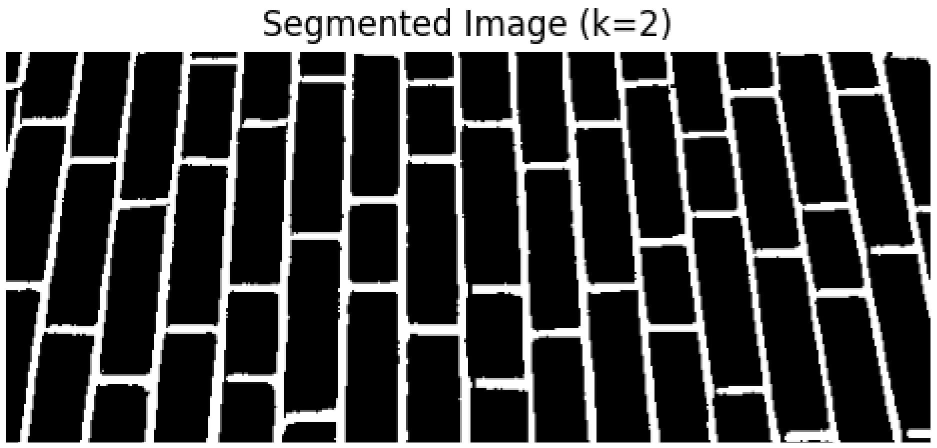
```
Peaks in the histogram:
Peak at intensity: 98
Peak at intensity: 119
```

```
    Peak at intensity: 122
```

3. Segment the image using a k-means clustering algorithm with k=2 and represent the result

```
from skimage import color
from sklearn.cluster import KMeans


# Reshape the image to a 2D array
height, width = image.shape[:2]
reshaped_image = image.reshape(height * width, -1)

# Perform k-means clustering with k=2
kmeans = KMeans(n_clusters=2, random_state=0)
kmeans.fit(reshaped_image)


#v_kmeans = kmeans.predict(reshaped_image)
#I_kmeans = np.reshape(v_kmeans,(image.shape[0],image.shape[1]))
# Get the labels assigned to each pixel
labels = kmeans.labels_

# Reshape the labels back to the original image shape
segmented_image = labels.reshape(height, width)

# Visualize the segmented image
plt.figure(figsize=(8, 6))
plt.imshow(segmented_image, cmap='gray')
plt.title("Segmented Image (k=2)")
plt.axis('off')
plt.show()
```

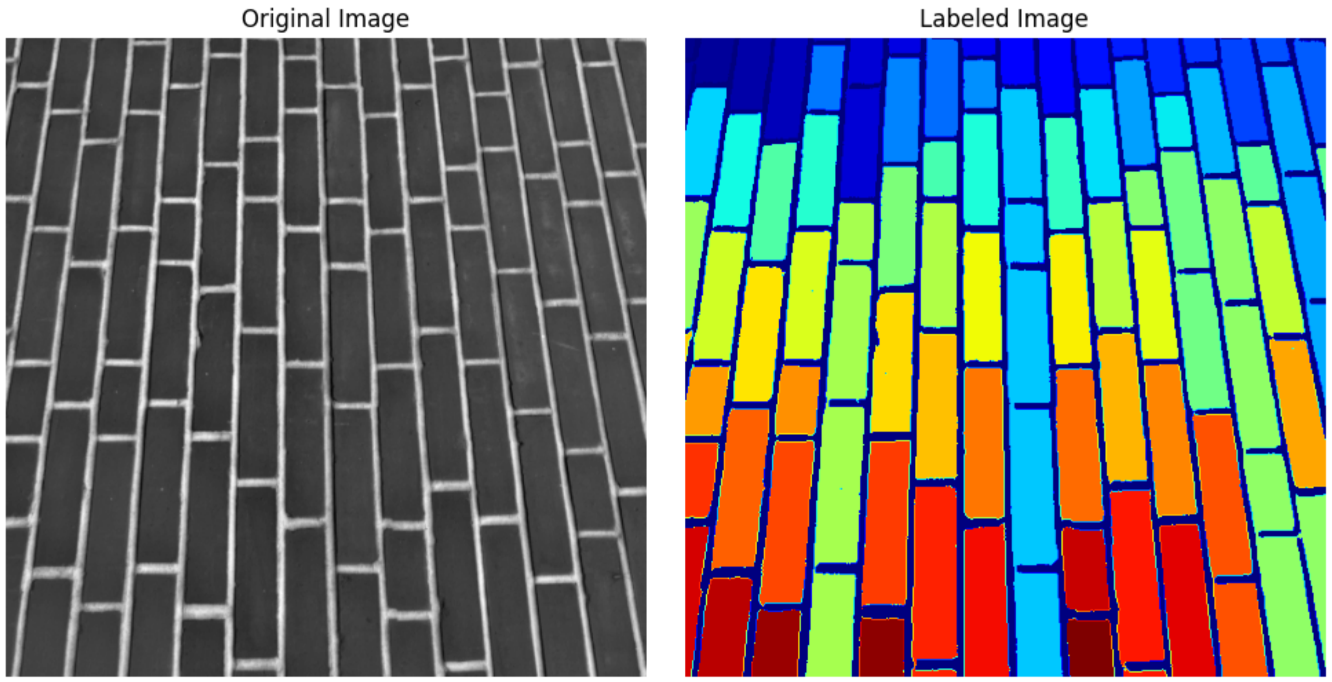
```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: T
  warnings.warn(
```



4. Label the objects found in the image. Generate a figure with the original image and the labeled image. How many bricks are there in the image?

```
import numpy as np
from skimage import data, measure
from skimage.measure import label
from skimage.filters import threshold_otsu

# Apply Otsu threshold for segmentation :
thresh = threshold_otsu(image)
bw = image <thresh # keep lighter regions with grayscale intensities above threshold

# Label the objects in the segmented image

labeled_image, nregions = label(bw,return_num=True)
#labeled_image = measure.label(segmented_image)

# Count the number of bricks

brick_count = np.max(labeled_image)

# Visualize the original image and the labeled image
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(image, cmap='gray')
axes[0].set_title('Original Image')
axes[0].axis('off')
axes[1].imshow(labeled_image, cmap=plt.cm.jet)
axes[1].set_title('Labeled Image')
axes[1].axis('off')
plt.tight_layout()
plt.show()
```

```
# Print the number of bricks
print('Number of bricks: {}'.format(brick_count))
```

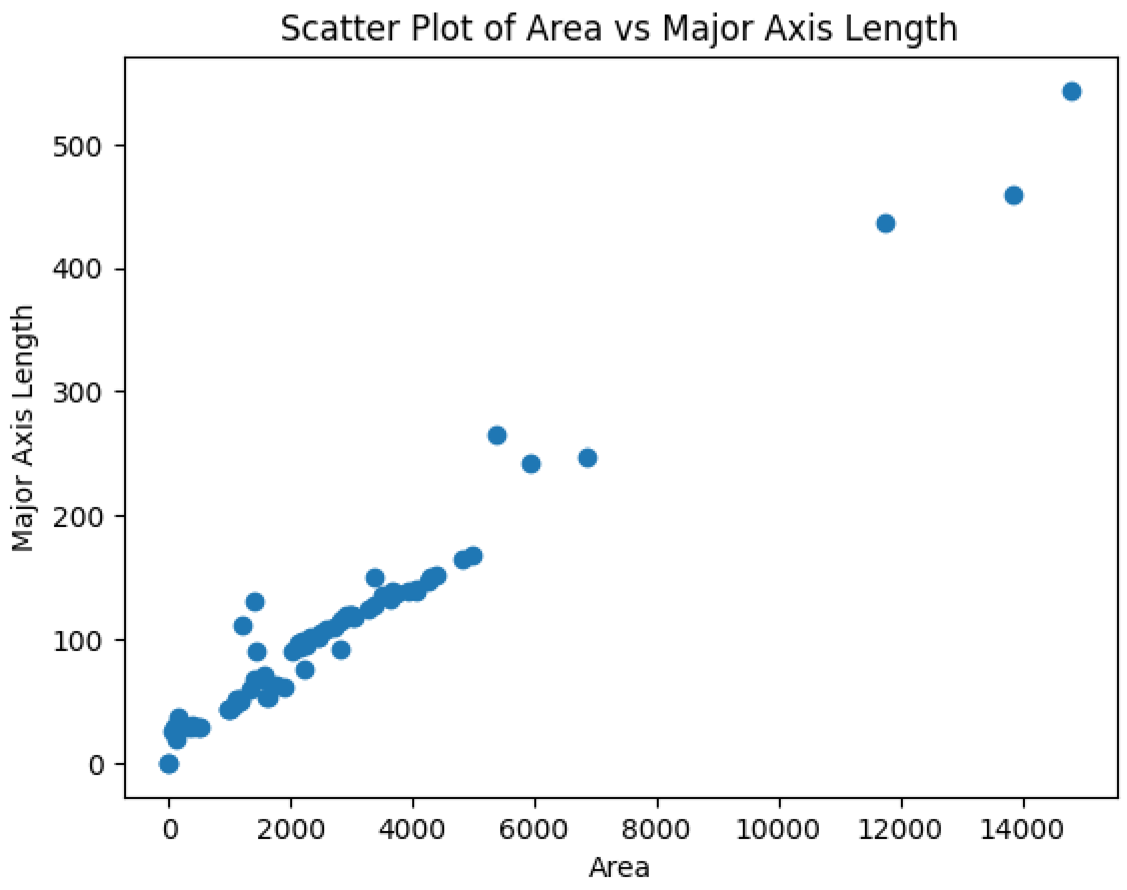


```
Number of bricks: 78
```

4. Extract the area, major axis length and orientation of each brick. Report the average values of the extracted features. Represent a scatterplot of the area vs major axis length.

```
# Calculate region properties for each labeled object
props = measure.regionprops(labeled_image)

# Extract the area, major axis length, and orientation of each brick
areas = [prop.area for prop in props]
major_axis_lengths = [prop.major_axis_length for prop in props]
orientations = [prop.orientation for prop in props]

# Calculate the average values of the extracted features
```

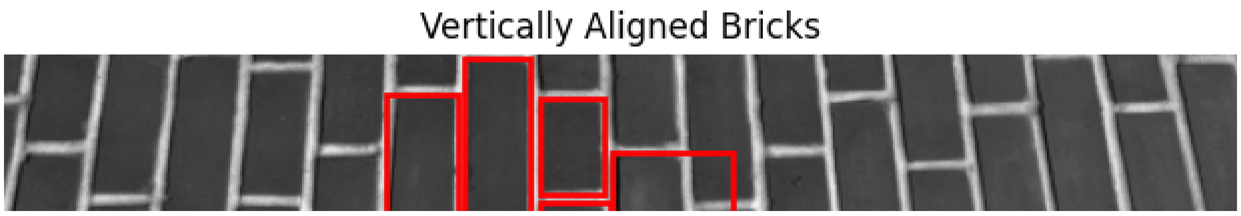
```
avg_area = np.mean(areas)
avg_major_axis_length = np.mean(major_axis_lengths)
avg_orientation = np.mean(orientations)

# Scatter plot of area vs major axis length
plt.scatter(areas, major_axis_lengths)
plt.xlabel('Area')
plt.ylabel('Major Axis Length')
plt.title('Scatter Plot of Area vs Major Axis Length')
plt.show()

# Print the average values of the extracted features
print('Average Area: {}'.format(avg_area))
print('Average Major Axis Length: {}'.format(avg_major_axis_length))
print('Average Orientation: {}'.format(avg_orientation))
```



```
Average Area: 2735.24358974359
Average Major Axis Length: 109.8645005736734
Average Orientation: 0.060453944913413014
```

5. Find the bricks that are vertically aligned (consider vertical bricks as the ones with an orientation between -0.05 and 0.05 degrees).

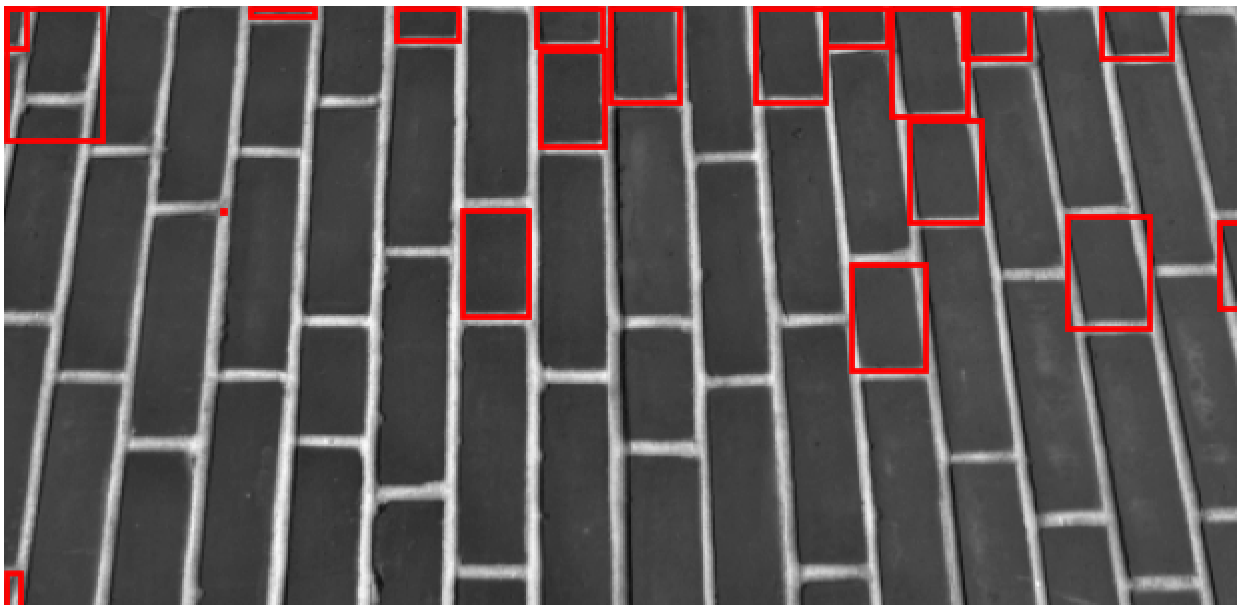
```
# Find vertically aligned bricks
vertically_aligned_bricks = []
for prop in props:
    orientation = prop.orientation
    if -0.05 <= orientation <= 0.05:
        vertically_aligned_bricks.append(prop)

# Plot the original image with labeled vertically aligned bricks
fig, ax = plt.subplots(figsize=(8, 8))
ax.imshow(image, cmap='gray')

for brick in vertically_aligned_bricks:
    minr, minc, maxr, maxc = brick.bbox
    rect = plt.Rectangle((minc, minr), maxc - minc, maxr - minr,
                          fill=False, edgecolor='red', linewidth=2)
    ax.add_patch(rect)

plt.title('Vertically Aligned Bricks')
plt.axis('off')
plt.show()
```



6. Find the bricks that have a length shorter than half the average length of the bricks (short bricks)

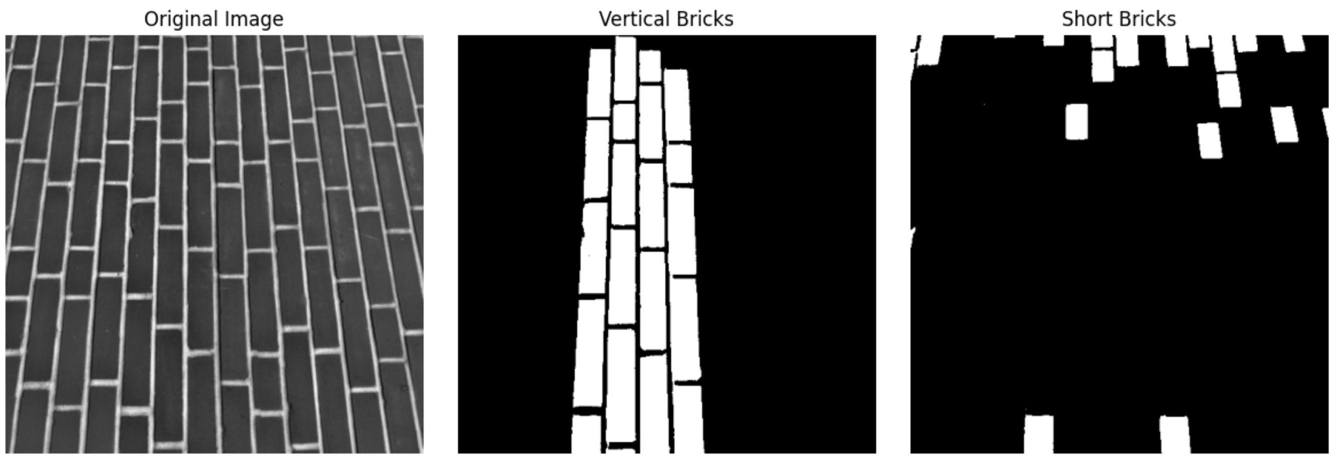
```
# Calculate the average length of the bricks
brick_lengths = [prop.major_axis_length for prop in props]
average_length = np.mean(brick_lengths)

# Find short bricks (length < half of the average length)
short_bricks = [prop for prop in props if prop.major_axis_length < average_length / 2]

# Plot the original image with labeled short bricks
fig, ax = plt.subplots(figsize=(8, 8))
ax.imshow(image, cmap='gray')

for brick in short_bricks:
    minr, minc, maxr, maxc = brick.bbox
    rect = plt.Rectangle((minc, minr), maxc - minc, maxr - minr,
                         fill=False, edgecolor='red', linewidth=2)
    ax.add_patch(rect)

plt.title('Short Bricks')
plt.axis('off')
plt.show()
```

Short Bricks



7. Generate a 3-panel figure with the original image, the image with the vertical bricks and an image with the short bricks.

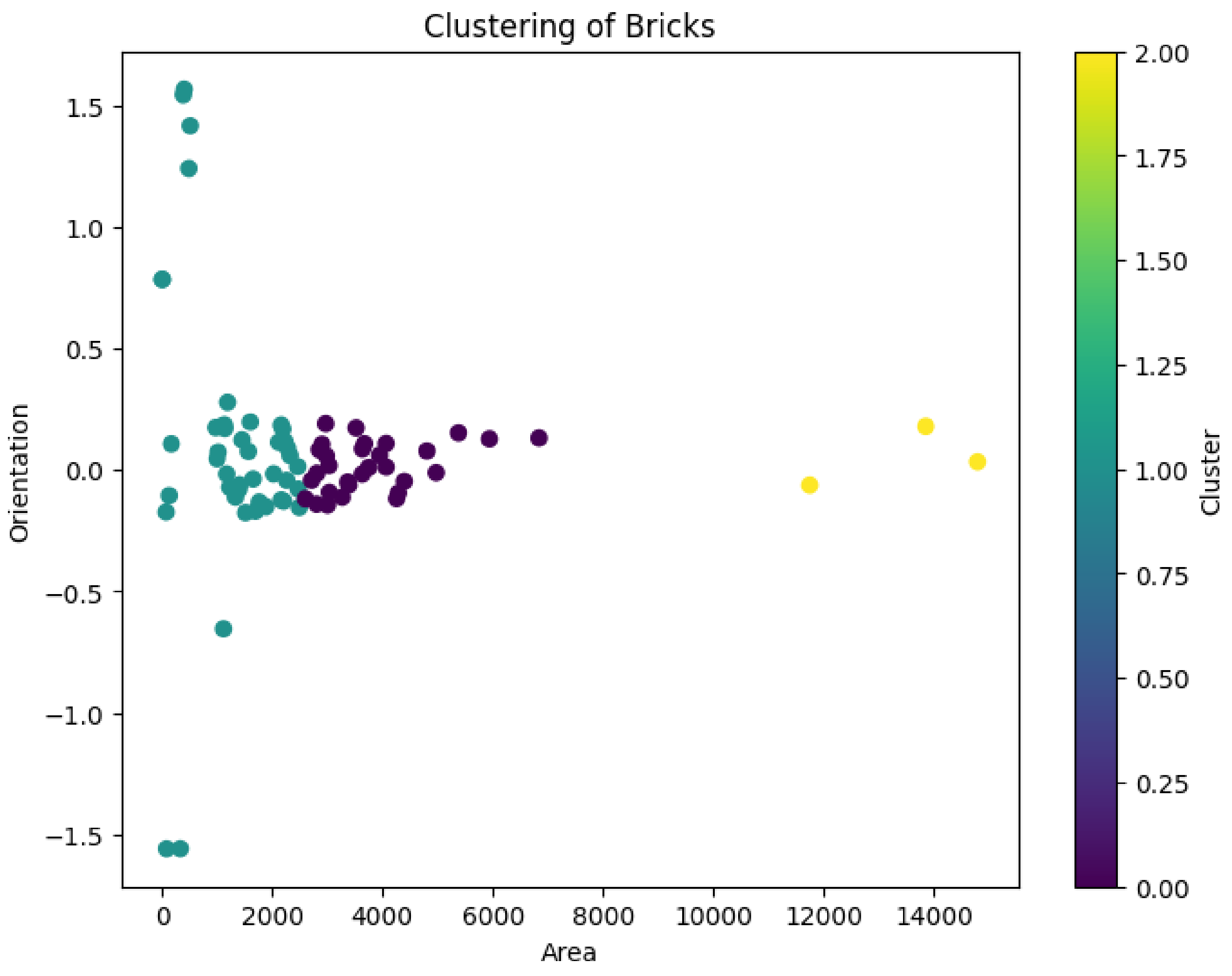
```
# Create a 3-panel figure
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

# Plot the original image
axes[0].imshow(image, cmap='gray')
axes[0].set_title('Original Image')
axes[0].axis('off')

# Plot the image with vertical bricks
image_vertical_bricks = np.isin(labeled_image, [brick.label for brick in vertically_aligned_b
axes[1].imshow(image_vertical_bricks, cmap='gray')
axes[1].set_title('Vertical Bricks')
axes[1].axis('off')

# Plot the image with short bricks
image_short_bricks = np.isin(labeled_image, [brick.label for brick in short_bricks])
axes[2].imshow(image_short_bricks, cmap='gray')
axes[2].set_title('Short Bricks')
axes[2].axis('off')

plt.tight_layout()
plt.show()
```



8. Apply k-means to the area and orientation features in order to cluster the bricks in three groups. Represent the result and provide an interpretation of the three types of bricks identified by the clustering algorithm.

```
# Extract area and orientation features from the bricks
areas = np.array([prop.area for prop in props])
orientations = np.array([prop.orientation for prop in props])

# Combine the features into a single array
features = np.column_stack((areas, orientations))

# Apply k-means clustering with k=3
kmeans = KMeans(n_clusters=3, random_state=0)
kmeans.fit(features)

# Get the cluster labels assigned to each brick
labels = kmeans.labels_

# Create a scatter plot of area vs major axis length
plt.figure(figsize=(8, 6))
plt.scatter(areas, orientations, c=labels, cmap='viridis')
plt.xlabel('Area')
plt.ylabel('Orientation')
plt.title('Clustering of Bricks')
plt.colorbar(label='Cluster')
plt.show()
```

```
    /usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: 
      warnings.warn(
```



9. Explain the different levels at which we can apply supervised classification algorithms to image processing

In image processing, supervised classification algorithms can be applied at different levels depending on the specific task and the nature of the data. For example:

Pixel-Level Classification: At the pixel level, each individual pixel in the image is classified based on its features or attributes. The goal is to assign a specific class or label to each pixel. This type of classification is useful for tasks such as image segmentation, where the objective is to partition the image into distinct regions or objects.

Object-Level Classification: At the object level, the classification is performed on complete objects or regions of interest within the image rather than individual pixels. The features used for classification are typically derived from the properties of the objects, such as shape, texture, or color. Object-level classification is often applied in tasks like object recognition or detection, where the goal is to identify and classify specific objects in the image.

Scene-Level Classification: At the scene level, the classification is performed on entire images or scenes. The features used for classification can include global characteristics of the scene, such as overall color distribution or texture patterns. Scene-level classification is commonly used for tasks like image categorization or scene understanding, where the goal is to classify images into broader categories or classes.

Hierarchical Classification: Hierarchical classification involves applying multiple levels of classification in a hierarchical manner. The classification starts at a coarse level, such as scene-level classification, and then progresses to finer levels, such as object or pixel-level classification. This approach allows for a more detailed and comprehensive analysis of the image data.

It's important to note that the choice of classification level depends on the specific problem and the available data. Different levels of classification provide different levels of granularity and can be used in combination to achieve more complex analysis and understanding of the image data.

10. Describe the main steps required to perform segmentation of an RGB image using an unsupervised clustering algorithm

The main steps to perform segmentation of an RGB image using an unsupervised clustering algorithm are as follows:

Preprocessing: If necessary, apply preprocessing steps to the RGB image, such as noise reduction, color space conversion, or normalization. This step helps to enhance the quality of the image and improve the performance of the clustering algorithm.

Feature Extraction: Extract relevant features from the RGB image that will be used by the clustering algorithm. Common features include color values (e.g., RGB or HSV channels), texture information, or spatial features. The choice of features depends on the specific segmentation task and the characteristics of the image.

Feature Vector Creation: Combine the extracted features into a feature vector representation for each pixel or region of interest in the image. The feature vector should capture the relevant information for clustering.

Clustering Algorithm Selection: Choose an appropriate unsupervised clustering algorithm for the segmentation task. Popular choices include K-means, Gaussian Mixture Models (GMM), or Hierarchical Clustering. Consider factors such as the desired number of segments and the algorithm's ability to handle high-dimensional feature vectors.

Clustering: Apply the selected clustering algorithm to the feature vectors to group pixels or regions into clusters. The algorithm assigns each pixel or region to a cluster based on the similarity of their feature vectors.

Postprocessing: Perform any necessary postprocessing steps to refine the segmentation result. This may include removing small or noisy clusters, smoothing the boundaries between segments, or merging similar segments.

Visualization: Visualize the segmented image to assess the quality of the segmentation result. This can be done by assigning different colors or labels to each segment and overlaying them on the original image.

Evaluation: Evaluate the segmentation result using appropriate metrics or qualitative assessments. This helps to assess the accuracy and effectiveness of the segmentation algorithm.

It's important to note that the specific implementation details and parameter settings may vary depending on the chosen algorithm and the characteristics of the image data. Experimentation and fine-tuning of parameters may be required to achieve optimal segmentation results.